Formal syntax for *Garter*, a tiny relative of Python

Susan Fox

This document lays out the formal lexical and syntactic structure of Garter, a Python-like language. You'll be writing a compiler for Garter this semester. You do not need to know Python in order to understand Garter.

1 Lexical structure

The lexical structure of a language is the form of individual symbols or tokens in the language: numbers, identifiers, syntax symbols, delimiters, and so on.

Garter is more simple than Python, in that it uses explicit block markers rather than whitespace indentation. Rather than using newline markers to end expressions, explicit markings will be used.

A program in Garter is made up of ASCII text. Whitespace and comments are ignored by the system. Whitespace includes spaces, tabs, and newlines (however these are represented on the particular computer). A comment begins with the special character `#` and continues to the end of the line. Otherwise, a program is made up of tokens, separated by either whitespace or, when unambiguous, written right next to each other. Tokens include syntax symbols, identifiers (both reserved words and variables), operator symbols, and data literals.

<token> ⟶ <identifier> | <number>
| `(` | `)` | `:` | `;` | `,` | `[` | `]` | `=`
| `+` | `-` | `*` | `/` | `%` | `**`
| `<` | `>` | `<=` | `>=` | `==` | `!=`

1.1 Identifiers

Identifiers are variables in Garter, and also a set of reserved words used for commands in the language. An identifier must start with an alphabetic character, and be made up of letters, digits, and the underscore. Identifiers in Garter are case sensitive: `myVar` is different from `Myvar`. Below are formal rules for legal identifiers.

<identifier> ⟶ <letter> (<letter> | <digit> | `_`)*
<letter> ⟶ `a` | `b` | ··· | `z` | `A` | `B` | ··· | `Z`
<digit> ⟶ `0` | `1` | `2` | ... | `9`

The reserved words in Garter may be recognized lexically or syntactically, below is the list.

`def` `enddef` `if` `elif` `else` `endif` `and` `or` `not` `for` `in` `endfor` `while` `endwhile` `print` `return` `True` `False`

1.2 Numbers

Garter includes only 32-bit integer numbers. The lexical rules don't specify the limit on how many digits an integer can have, but that is a semantic rule we'll need to enforce.

<number> ⟶ <integer> *add real here if you wish*
<integer> ⟶ <digit>$^+$

1.3 Delimiters

Some tokens have fixed length, like <=, for example. Others, like numbers and identifiers, have variable length. Certain symbols called delimiters automatically tell us that a variable-lengthed token has ended, and a new token is about to start. Whitespace is the most obvious delimiter. In Garter, the following characters also serve as delimiters, in addition to whitespace. Some of the following are only delimiters in certain contexts (= , for example, is a delimiter except when it is a part of a multi-character token). Other characters may serve as delimiters in other contexts (a character following a number, for example).

```
( ) : ; , + - * / % [ ] = < > ! #
```

2 Syntactic structure

The grammar below describes what the parser must recognize as a valid Garter program. It is organized conceptually, rather than in the most efficient manner for parsing the language. Your first task in creating a parser should be to collect together, simplify, and reorganize the grammar rules. Be aware that some terms on the right-hand side of rules refer to lexical rules, which will be handled by the scanner (also called the lexical analyzer). Also be aware that some terms may be used after they are defined.

A program in Garter is a sequence of statements, and function definitions. Python permits simple expressions that evaluate to a value to be at top-level in a program. For simplicity of parsing, I've removed it. You may add it back if you really want it. Running a program means executing each item in the sequence in turn. Function definitions create functions that can be called later and statements are executed.

<program> ⟶ (<statement> | <func def>)$^+$

2.1 Statements

Statements are pieces of a program that change the state of the program. They give a command to the computer to do something, rather than having a value. Statements in Garter are either simple statements (assignment statements, etc.) or compound statements that specify control structions. Sequences of statements occur in most of the compound forms. In Garter, statements have no explicit separator symbol, but are simple enough that it is still parsable.

<statement> ⟶ <simple stmt> `;` | <compound stmt>
<stmt seq> ⟶ (<statement>)$^+$

Simple statements include assignment, print, and return statements. Assignment statements must have an identifier or a subscripted identifier on the lefthand side, and then a single expression on the righthand side. Print statements can have nothing following the keyword, or a sequence of expressions, separated by commas. Return statements must be followed by a single expression.

<simple stmt> ⟶ `pass` | <assignment stmt> | <print stmt> | <return stmt>
<assignment stmt> ⟶ <target> = <expression>
<target> ⟶ <identifier> | <subscription>
<print stmt> ⟶ `print` <expr seq>
<return stmt> ⟶ `return` <expression>

Compound statements are control structures, if statements, or loops like while or for. Since Garter does not require rigid rules of indentation, each compound statement has an explicit `end` marker.

<compound stmt> ⟶ <if stmt> | <while stmt> | <for stmt>

If-then-else statements are notorious for being ambiguous in many programming languages. This grammar tries to avoid ambiguity by requiring an end marker for the if statement. Any number of else-if clauses are permitted, and the else clause is optional, but the end of an if statement must be marked by an `endif` keyword.

<if stmt> ⟶ `if` <expression> `:` <stmt seq> <elif clauses> `endif`
| `if` <expression> `:` <stmt seq> <elif clauses> `else` `:` <stmt seq> `endif`
<elif clauses> ⟶ (`elif` <expression> `:` <stmt seq>)*

A while loop checks its test expression each time through the loop to see whether to continue the loop or not. Note that Garter loops do not have break or continue operations, you need to use conditional forms to end a loop, unless you are using the `return` statement.

<while stmt> ⟶ `while` <expression> `:` <stmt seq> `endwhile`

Garter's for loop, like Pythons, is an iterating form, where you specify list and the loop iterates over the members of it. As a matter of semantics, the expression in the loop must evaluate to a list.

<for stmt> ⟶ `for` <identifier> `in` <expression> `:` <stmt seq> `endfor`

2.2 Expressions

Expressions are pieces of a program that have a value. In Python, expressions are considered valid programs in and of themselves. Interactively, the value of the expression is printed, but when compiling and running a whole program some systems print the values of expressions, and others do not. Garter expressions are simplified, but similar.

The grammar rules below lay out the precedence rules of Garter: or expressions bind most weakly, followed by and, not, comparison operators, and then the arithmetic operators. The "power" operation (exponentiation) binds most strongly of the arithmetic operations.

<expression> ⟶ <or expr>
<or expr> ⟶ <and expr> | <or expr> `or` <and expr>
<and expr> ⟶ <not expr> | <and expr> `and` <not expr>
<not expr> ⟶ <comparison> | `not` <not expr>
<comparison> ⟶ <add expr> | <add expr> <comp operator> <add expr>
<comp operator> ⟶ `<` | `>` | `<=` | `>=` | `==` | `!=` | `in` | `not in`

<add expr> ⟶ <mult expr> | <add expr> + <mult expr> | <add expr> − <mult expr>
<mult expr> ⟶ <unary expr> | <mult expr> `*` <unary expr>
| <mult expr> `/` <unary expr> | <mult expr> `%` <unary expr>
<unary expr> ⟶ <power expr> | − <unary expr> | + <unary expr>
<power expr> ⟶ <primary> `**` <unary expr> | <primary>

Primaries are expressions with the highest precedence in grouping. Simple atoms, subscripting of lists, and procedure calls are all primaries. Atoms are the most simple kind of expression. An atom is a number, or an identifier, or a parenthe¡sized expression, or a list.

In a subscript expression, there are two parts, another primary and an expression. The primary must evaluate to a list, and the expression must be an integer. These are issues of semantics, not syntax, though.

A procedure call in Python allows any expression that evaluates to a "callable object" to be the first part of a call. In Garter, however, it must be an identifier, and semantically it must be the name of a known procedure. Also semantically, the number of expressions in the expression sequence must equal the number of expected parameters of the procedure.

<primary> ⟶ <atom> | <subscription> | <call>
<atom> ⟶ <number> | <identifier> | <paren expr> | <list expr>
<paren expr> ⟶ `(` <expression> `)`
<list expr> ⟶ `[` <expr seq> `]`
<expr seq> ⟶ (<expression> (`,` <expression>)*)?
<subscription> ⟶ <identifier> (`[` <expression> `]`)+
<call> ⟶ <identifier> `(` <expr seq> `)`

Note: the <expr seq> *rule may look a bit odd. It means that the whole thing is optional: the empty string is a valid* <exprseq> *. If not empty, then it may be a single expression, followed by zero or more instances of a comma followed by another expresion.*

2.3 Function definitions

A function is a piece of code that takes standard inputs and produces some result. Garter uses an explicit end marker to simplify the parsing process.

<func def> ⟶ `def` <identifier> `(` <parameters> `)` `:` <stmt seq> `enddef`
<parameters> ⟶ ϵ | <param list>
<param list> ⟶ <identifier> | <identifier> `,` <param list>